Project: Emulate a Gameboy

Objective

Write a Nintendo Gameboy emulator. Your finished project should play *Supermarioland*, as well as the test ROM images you are given.

Partners

You may work on this project individually or in groups of two.

Software

You have two options, with slightly different steps depending on which you choose.

1. You use Qt Creator, installed from

https://www.qt.io/download

Make sure you download the open source version, *not* the "free trial" version. You'll have to read through the page carefully to find it.

The advantage of Qt creator is that it gives you a development environment and graphics libraries so you can make your gameboy run in a window. The downside is that many students have trouble installing it, especially on some newer laptops.

2. You use a server set up for this class. I will provide instructions in class on how to use this. The downside is that your project will use ASCII art.

Resources

You are given the following:

Z80.cpp / Z80.h	A Z80 CPU emulator
	Test ROM image

PART 1. Try out the Z80

Here is a simple test program that sums 6+5+4+3+2+1 = 21.

```
void dosum()
{
        int b=6;
    int a=0;

    while(b>0)
    {
        a=a+b;
        b=b-1;
```

```
    }
    halt;
}
```

In assembly, this is the equivalent program.

```
0    ld    b,6   06 06
2    ld    a,0   3E 00
4    add   b     80
5    dec   b     05
6    jg    L2    C2 04 00
9    halt                76
```

You can store this in a single global array as follows:

```
char rom[]={0x06,0x06,0x3e,0x00,0x80,0x05,0xc2,0x04,0x00,0x76};
```

Write *gameboy.cpp* as follows:

1. Put in the rom[] array and the following two functions:

```
unsigned char memoryRead(int address) {  }

void memoryWrite(int address, unsigned char value) { }
```

2. Inside memoryRead, use *address* to read from the rom array and return the appropriate byte.

3. In *main(),* construct a Z80 as follows:

```
z80 = new Z80 ( memoryRead, memoryWrite);
```

4. Call *reset()* on the z80 object, then set its PC variable to 0 (the beginning of the rom code).

5. Call *doInstruction()* on the z80 object, and print out its PC, instruction, A, and B variables. You should see it complete the first ***ld b,6*** instruction, PC should be 2, and B should be 6.

6. Put *doInstruction()* in a while loop, and stop the while loop when z80->halted is *true.*

Compile and run the program. On the server, type:

g++ -o gameboy gameboy.cpp Z80.cpp

If you're using Qt Creator, make a non-Qt C++ project, and copy over the appropriate files.

When you run it, you should see it run a couple dozen instructions and stop. At the end, A should hold 21.

Now we'll run the same program, but we will read it from a file instead of hardcode it in an array.

7. Make a global *rom* array *char* rom* and integer *romSize*. Don't bother initializing them.

8. Before constructing the z80 object, read from the file *testrom.gb* and copy it character by character into your rom array:

```
ifstream romfile("testrom.gb", ios::in|ios::binary|ios::ate);
streampos size=romfile.tellg();

rom=new char[size];

romSize=size;

romfile.seekg(0,ios::beg);

romfile.read(rom,size);

romfile.close();
```

Don't initialize PC to 0 any longer.

Verify that it still prints out A=21.

PART 2: Displaying tile graphics

The gameboy screen looks the best using the Qt graphics library. However, many students have difficulty installing Qt creator or getting it to run Qt graphics.

QT version

You are provided a file *screendump* that holds a video memory snapshot as a series of integers. In this part, you will make a video memory, load this snapshot into it, and render that image in a window.

You can now include files *gameboy.pro*, *screen.cpp*, and *screen.h* in your project. Call qmake and make to compile everything together.

Make sure your *main()* is now written as

```
extern QApplication* app;

int main(int argc, char* argv[])
{
    setup(argc,argv);

      //part 1 code here

      //part 2 code here

    app->exec();
}
```

An empty window should show up, and your program from Part 1 should run as before.

Non-Qt version

A template *gameboy_noQt.cpp* is provided for you. Do not use the gameboy.pro or screen files. You will put all your code in this step into function *renderScreen* where it tells you to insert code.

1. Make some global variables to hold the video memory:

```
unsigned char graphicsRAM[8192];
int palette[4];
int tileset, tilemap, scrollx, scrolly;
```

In main(), read the first 8192 integers from *screendump* into graphicsRAM.

```
int n;
ifstream vidfile("screendump.txt",ios::in);
for(int i=0; i<8192; i++){
        int n;
```

```
            vidfile>>n;

            graphicsRAM[i]=(unsigned char)n;
    }
```

Then read the other variables:

```
vidfile >> tileset;
vidfile >> tilemap;
vidfile >> scrollx;
vidfile >> scrolly;


vidfile >> palette[0];
vidfile >> palette[1];
vidfile >> palette[2];
vidfile >> palette[3];
```

2. Your task now is to calculate a color value 0, 1, 2, 3 for each of the 160x144 pixels on the gameboy screen. Call function *updateSquare(x, y, color)* to paint the appropriate square on the window.

Read *http://imrannazar.com/GameBoy-Emulation-in-JavaScript:-Graphics* to see how the gameboy converts video memory into an image. Briefly, it works like as follows. Given a pixel *x, y*, use the following steps to figure out its color:

First apply a *scroll* to shift the image by some amount. Let $x = (x + scrollx) \& 255$ and $y = (y + scrolly) \& 255$

Next figure out which tile the pixel x,y belongs to. The 160 x 144 screen is divided into 8x8 tiles. Given a pixel *x, y*, the tile coordinate is *tilex = x/8 , tiley = y/8*.

These tiles are organized in rows from left to right and then from top to bottom. There are 32 tiles in each row -- more than can be seen on the screen. The list of tile numbers is called the "tile map". The tile's position in the map is *tileposition = tiley * 32 + tilex*

There are two tile maps. Map 0 is located in the graphicsRAM at address 0x1800 (6144), Map 1 is located at address 0x1c00 (7168). The map actually used depends on the value of the *tilemap* variable. The tile index is thus:

- if tilemap is 0, *tileindex = graphicsRAM[0x1800 + tileposition*]
- if tilemap is 1, *tileindex = graphicsRAM[0x1c00 + tileposition]*

The tile encoding itself -- the pixel colors in the tile -- is located in the bottom of graphicsRAM. Every tile entry is 16 bytes in size. The location of the entry, based on the index, is different for tilesets 0 and 1.

If the tileset is 1, the address is simply: *tileaddress = tileindex * 16*
If it is 0, however, calculating the address becomes more complicated. The tile indices are treated as a signed number, with negative numbers occuring below 0x1000 and positive above 0x1000. The complete formula for tilemap 0 becomes:

if tileindex >= 128, tileindex = tileindex - 256
*tileaddress = tileindex * 16 + 0x1000*

Each 8x8 tile is encoded as 8 rows, each row consisting of two bytes. To get the two bytes for the pixel we want within the tile, calculate *xoffset = x % 8* and *yoffset = y % 8* The two bytes -- we'll call them *row0* and *row1* -- are found at

*row0 = graphicsRAM[tileaddress + yoffset * 2]*
*row1 = graphicsRAM[tileaddress + yoffset * 2 + 1]*

Now for the hardest part. Each bit within the row byte corresponds to a pixel within the row. For example, a row of 8 pixels might look like:

3 2 0 0 1 2 3 3

In binary, 0 is 00, 1= 01, 2=10, 3=11. This sequence becomes:

11 10 00 00 01 10 11 11

Now split this into two bytes. The first byte gets the lower bits, the second gets the higher bits:

row0 = 1 0 0 0 1 0 1 1 = 139
row1 = 1 1 0 0 0 1 1 1 = 199

The following from *http://fms.komkon.org/stuff/gameboy.faq* shows how this works across an entire tile:

```
Tile:                          Image:

.33333..                        .33333.. -> 01111100 -> 7Ch
22...22.                              01111100 -> 7Ch
11...11.                       22...22. -> 00000000 -> 00h
2222222. <-- digits represent                11000110 -> C6h
33...33.    color numbers       11...11. -> 11000110 -> C6h
22...22.                              00000000 -> 00h
11...11.                       2222222. -> 00000000 -> 00h
.......                              11111110 -> FEh
                            33...33. -> 11000110 -> C6h
                                     11000110 -> C6h
                            22...22. -> 00000000 -> 00h
                                     11000110 -> C6h
                            11...11. -> 11000110 -> C6h
                                     00000000 -> 00h
                            ........ -> 00000000 -> 00h
                                     00000000 -> 00h
```

So how do you get pixel x's color? Use the bit shifting operator >> to push all the bits to the right. Then use bitwise AND (&) to zero out all the bits except the one you're interested in.

Example: Suppose *xoffset* is **1**, meaning we want the second pixel from the left (in bold)

*row0 = 139 = 1 **0** 0 0 1 0 1 1*
*row0shifted = row0 >>(7 - **1**) = 1 **0***

row0capturepixel = row0shifted & 1 = **0**

Now do this again for row1:

row1 = 199 = 1 ***1*** *0 0 0 1 1 1*
row1shifted = row1 >> (7 - ***1****) = 1* ***1***
row1capturepixel = row1shifted & 1 = ***1***

Put them together:

*pixel = row1capturepixel * 2 + row0capturepixel =* ***1*****2 +* ***0*** *=* ***2***

You're not completely done yet. *pixel* doesn't hold the actual color, but an index to a "palette" or array of colors. Your final color is:

color = palette [pixel];

Now apply this to every pixel on the screen:

```
for(int column=0; column<160; column++)
    for (int row = 0; row<144; row++)
                int x=column, y=row;
                //do the above calculations to get color
                updateSquare(column,row,color);
    onFrame();
```

If the test image of a house surrounded by trees shows up, congratulations! Now put those for loops and computation into a function *void renderScreen();*

PART 3: Memory and Timing

In this step we will connect your work from the first two parts by adding on memory management and video timing. When you finish this part you will be able to load and partially run several games on your emulator.

Step A. Memory Map

You should now alter your *memoryRead* and *memoryWrite* functions so that they not only access the rom but several other memory units as well. Based on the range of the *address*, you should implement the following memory layout:

Address range (in hex)	memoryRead returns:	memoryWrite sets this to value
0 - 0x3FFF	rom[address]	*nothing*
0x4000 - 0x7FFF	rom[romOffset+address%0x4000]	*nothing*
0x8000 - 0x9FFF	graphicsRAM[address%0x2000]	graphicsRAM[address%0x2000]
0xC000 - 0xDFFF	workingRAM[address%0x2000]	workingRAM[address%0x2000]
0xFF80 - 0xFFFF	page0RAM[address % 0x80]	page0RAM[address % 0x80]
0xFF00	0xf	keyboardColumn
0xFF40		setControlByte(value)
0xFF41	getVideoState()	videostate
0xFF42	scrolly	scrolly
0xFF43	scrollx	scrollx
0xFF44	line	line
0xFF45	cmpline	cmpline
0xFF47		setPalette(value)
anything else	*nothing - just return 0*	

Now add the following global variables:

```
int HBLANK=0, VBLANK=1, SPRITE=2, VRAM=3;
unsigned char workingRAM[0x2000];

unsigned char page0RAM[0x80];

int line=0, cmpline=0, videostate=0, keyboardColumn=0, horizontal=0;
int gpuMode=HBLANK;
int romOffset = 0x4000;
long totalInstructions=0;
```

And functions:

```
void setControlByte(unsigned char b) {
      tilemap=(b&8)!=0?1:0;
      tileset=(b&16)!=0?1:0;
 }
void setPalette(unsigned char b) {
        palette[0]=b&3;
        palette[1]=(b>>2)&3;
```

```
        palette[2]=(b>>4)&3;
        palette[3]=(b>>6)&3;
}

unsigned char getVideoState() {
      int by=0;
      if(line==cmpline) by|=4;
      if(gpuMode==VBLANK) by|=1;
      if(gpuMode==SPRITE) by|=2;
      if(gpuMode==VRAM) by|=3;
      return (unsigned char)((by|(videostate&0xf8))&0xff);
 }
```

Make sure your program still compiles and runs without segmentation faulting.

Step B. Video Timing

Now you will expand your main loop to include video timing.

Previously, your main loop looked like this:

```
while(true)
        z80->doInstruction();
        if(z80->halted) break;
```

Now you should change this to do the following:

while true:
    if not halted, do an instruction
    check for and handle interrupts
    figure out the screen position and set the video mode
    redraw the screen

Check for and handle interrupts:

When an interrupt comes in from the screen the Z80 doesn't immediately respond but waits for a few instructions. The following code will delay and call an interrupt at the correct time.

```
if(z80->interrupt_deferred>0)
{
        z80->interrupt_deferred--;
        if(z80->interrupt_deferred==1)
        {
                z80->interrupt_deferred=0;
                z80->FLAG_I=1;
        }
}
z80->checkForInterrupts();
```

Figure out screen position and set the video mode

Recall that the gameboy video is handled by a beam moving row by row, from left to right. Two variables determine what position on the screen is currently being drawn: *horizontal* and *line*. You will

use the total number of instructions to update these variables.

First keep track of the total instructions. Put *totalInstructions++* at the end of your loop.

Now determine your horizontal position. Given that the gameboy runs approximately 61 instruction for each row of the screen, you can say:

horizontal = (int) ((totalInstructions+1)%61);

Now set your *gpuMode* variable using *horizontal* and *line*, as follows:

1. if *line* is 145 or bigger, your gpuMode is VBLANK
2. otherwise, if *horizontal* is less than or equal to 30, your mode is HBLANK
3. otherwise, if *horizontal* is between 31 and 40 (inclusive), your mode is SPRITE
4. otherwise your mode is VRAM

Check if your *horizontal* is 0. If so, do the following steps:

1. Increment *line*
2. If *line* is 144, call *z80->throwInterrupt(1)* to signify that you've reached the VBLANK state
3. There is a special variable *cmpline* that gameboy programs use to wait until a certain line is reached. If *line%153==cmpline* and *(videostate&0x40)!=0* , then call *z80->throwInterrupt(2)*
4. If *line* is 153, set *line* back to 0, and call your *renderScreen* function that your wrote in Part 2.

Don't call app->exec() anymore. Just make sure you call *onFrame()* in your renderScreen function.

You can now try loading and running the games *opus5.gb* and *ttt.gb*. Although they won't respond to keystrokes, you should see the games start on your emulator.

Rendering on every line

Some games, such as SuperMarioLand, will not work correctly unless video rendering is done after each line. Make the following changes:

1. Change your renderScreen function so that it takes the *row* as a parameter:
   int renderScreen(int row)
   now remove the for(int row=0; inner loop.

2. Remove the *onFrame* call from the end of your renderScreen function.

3. In main, don't call renderScreen when line==153. Instead call *onFrame*.

4. Now, every time you increment line, do the following:

```
if(line>=0 && line<144)
        renderScreen(line);
```

PART 4: ROM Banks

Most gameboy games more advanced than Tetris require more memory than there are addresses available. They accomplish this by dividing the ROM into "banks" which are swapped in and out of memory while the game is running. Because this is done on the cartridge, rather than in the gameboy itself, there are several different approaches used.

Early gameboy games, including SuperMarioLand, use a mode known as "memory bank control 1". In this mode, the bank is chosen by memory writes to ROM. A bank switch involves a write of the lower five bits of the bank to address 0x2000 and the upper two bits to address 0x4000.

First we need to know what type of cartridge is running. Create some new variables: *rombank*, *cartridgetype*, and *romsizemask*. After reading in the game file, set these variables as follows:

rombank should be set to 0
cartridgetype should be set to *rom[0x147]&3*
romsizemask should be set to one of the following values as indexed by the code in *rom[0x148]*
[0x7fff,0xffff,0x1ffff,0x3ffff,0x7ffff,0xfffff,0x1fffff,0x3fffff,0x7fffff]

Now, when the program writes to memory we will adjust romOffset accordingly. Notice that when romOffset is changed, memory reads between addresses 4000 and 8000 will be shifted to a different place in ROM.

In memoryWrite, check if the cartridgetype is 1, 2, or 3. If it is, then check the address. If it is between 2000 and 3fff,

remove the top bits of value and set it to 1 if it's 0:

```
value = value & 0x1f
if value==0
    value=1
```

zero out the low bits of rombank and replace it with value:

```
rombank = rombank & 0x60
rombank += value
```

compute a new offset

```
romOffset = (rombank*0x4000) & romsizemask
```

Alternatively, if the cartridgetype is 1, 2, or 3, and the address is between 4000 and 5fff:

get the last two bits of value: value=value&3
zero out the top bits of rombank and replace it with value

```
rombank=rombank&0x1f
rombank |= value<<5
```

compute a new offset

```
romoffset = (rombank*0x4000) & romsizemask
```

Now test this with mario.gb and ffl.gb. You should be able to run these games and advance through them, albeit without seeing many of the game components.

PART 5: Keyboard

Qt version

1. In screen.cpp two function calls *keydown* and *keyup* are commented out. Write these two functions in your gameboy.cpp and uncomment the colls in screen.cpp. For now, have those functions simply print out (cout) the integer value coming in.

Non-Qt version

1.
At the beginning of main() call function set_conio_mode();

At the beginning of main, make an int keyDownOrUp=0;

Next to where you call renderScreen in main, write:

```
if (kbhit()==1)
{
        int keycode=getch();
        if(keycode==3) return;
        if(keyDownOrUp==0)
                keydown(keycode);
        else
                keyup(keycode);
        keyDownOrUp=1-keyDownOrUp;
}
```

Now you can write the two functions *keydown* and *keyup*.

Both Qt and Non-Qt

2. Your gameboy needs eight keys to operate: left, right, down, up, A, B, START, SELECT. Choose keys on your keyboard to represent them. Find out what their keycodes are by pressing them and seeing what numbers are printed.

3. Make two variables *keys1* and *keys0* and initialize them to 0xf. Based on the key press, set them as follows:

pressing a key:

right	keys1 &= 0xe
left	keys1 &= 0xd
up	keys1 &= 0xb
down	keys1 &= 0x7
A	keys0 &= 0xe
B	keys0 &= 0xd
SELECT	keys0 &= 0xb
START	keys0 &= 0x7

releasing a key:

```
right   keys1 |= 1
left    keys1 |= 2
up      keys1 |= 4
down    keys1 |= 8

A       keys0 |= 1
B       keys0 |= 2
SELECT      keys0 |= 4
START       keys0 |= 8
```

The gameboy can only read either keys0 or keys1 at a time. Make a variable keyboardColumn=0. Then, in memorywrite:

```
if(address==0xff00)
    keyboardColumn = value
```

Now in memoryread:

```
if(address==0xff00)
    if ((keyboardColumn&0x30) == 0x10)
        return keys0
    else
        return keys1
```

Finally, when a key is pressed or released, throw an interrupt 0x10

Now try out opus5.gb and you should see the background scroll as you press a direction key. You can also try tetris.gb and be able to stack up blocks. The tetris blocks and the opus5 spaceship remain invisible, however.

PART 6: Sprites

A sprite is a graphical object that is drawn separately from the background tiles. Game characters, like Mario, or objects, such as mushrooms, typically appear as sprites.

Step 1: variables

First you will need some new variables. Create an unsigned char array *spriteRAM* of size 0x100 and two new palette arrays *objpalette0* and *objpalette1*, both of size 4. Update your memoryRead and memoryWrite functions so that memory reads and writes between 0xfe00 and 0xfe9f go to the spriteRAM. Additionally, in memoryWrite, writes to 0xff48 and 0xff49 should write to objpalettes 0 and 1, respectively. Follow the same method you used for writes to 0xff47.

Step 2: DMA

The gameboy has a DMA module that instantly copies from memory to sprite memory. Create a

function *dma* as follows:

```
void dma(int address) {
        address=address<<8;
        for(int i=0; i<0xa0; i++)
                memoryWrite(0xfe00+i,memoryRead(address+i));
}
```

Now go to memoryWrite. If the address is 0xff46, call *dma(value);*

Step 3: Rendering

Your next step should be to render the sprites. Do this in your renderScreen function in the innermost for loop, after you've determined the pixel's color, but before you call updateSquare.

spriteRAM holds a table of 40 sprites. Not all of them will be visible at the same time, but nevertheless, since they move independently, you will need to loop through all 40 and see if any are active and located at pixel (*column,row)*.

Write a for loop with iterator *spritenumber* and go through 40 sprites.

Each sprite entry contains four bytes:

0. the sprite's y position
1. the sprite's x position
2. the sprite tile's number within the tile set
3. options about the sprite

1. Get the y and x of the upper left corner of the sprite:
   spritey = spriteRAM[spritenumber*4+0]-16
   spritex = spriteRAM[spritenumber*4+1]-8

2. Sprites are 8x8. Check if the sprite covers your pixel's column and row:
   if spritex + 8 <= column or spritex > column ?
      don't bother with this sprite

   Do the same check for spritey and row.

3. Get the options bit. The top most bit tells whether the sprite is above or below the background. If (options&0x80)!=0 don't bother with the sprite.

Now we know we need to draw this sprite. This involves getting the sprite's tile out of the graphics RAM.

4. Get the sprite's tile number. the address of the sprite's bitmap within graphicsram will then be:
   tileaddress = tilenumber * 16

5. Get the x and y location within the sprite's tile. Use offsetx = column-spritex and offsety=row-spritey. There's catch, however. The sprite can be flipped horizontally or vertically. Look at options bit 6. If (options&0x40)!=0, you'll need to set offsety = 7-offsety. Do the same for (options&0x20)!=0

and offsetx.

6. You can now use the exact same approach for getting the sprite pixel as you did for getting a tile pixel. Get the row0, row1 bits and put them together to form a palette index.

7. If the palette index is 0, don't draw the sprite. Otherwise...

8. Use options bit 4 to choose one of the object palettes. if ((options&0x10)==0) use objpalette0, otherwise objpalette1. Use the palette index to get a color. This color will be what you will now pass to *updateSquare*.

Grading

Part 1	50 %
Part 2	60 %
Part 3	70 %
Part 4	80 %
Part 5	90 %
Part 6	100 %

Submission

By the deadline, you should submit on Blackboard:

- all your .cpp and .h files
- a zip file of your project directory
- a statement explaining what you did, what you attempted, and how far you got